Predicting Image Social Tags Using a Convolutional Neural Network

Panu Asikanius

University of Tampere
Faculty of Natural Sciences
Master's Degree Programme in Software Development
Master’s thesis
Supervisor: Martti Juhola
October 2018

University of Tampere
October 2018
Master's Degree Programme in Software Development
Panu Asikanius: Predicting Image Social Tags Using a Convolutional Neural Network
Master’s thesis, 52 pages
October 2018

Convolutional Neural Networks (CNN) are regarded as the state-of-the-art approach in image classification tasks, i.e. predicting appropriate labels for an image. Active ongoing research in the field has produced methods that rival human accuracy in image recognition. At the same time, the popularity of social networking sites has led to a huge number of user-uploaded photographs labeled with social tags or *#hashtags*.
This thesis provides an introduction to deep artificial neural networks, specifically when employed for the task of image recognition. The learnings are then applied to the development of a CNN trained on images and social tags collected from social networking sites. The goal is to predict relevant social tags for a newly uploaded photograph.
An evaluation of the proposed implemented approach reveals that it is indeed feasible to use a CNN to predict relevant tags for an image, but reaching high accuracy metrics is difficult. Further research is needed to improve the quality of the predictions.

Keywords: deep learning, machine learning, neural networks, image recognition, social tags

Acknowledgements

I would like to express my sincere gratitude to everyone who has helped me complete this thesis. First of all I want to thank my thesis supervisor *Martti Juhola* for his help. I also want to offer my thanks to the company I work for, *Vincit*, for enabling me to support myself financially while studying and for offering an inspiring and educational work environment for a student software developer.

I also want to thank my friends and family for all their help and support during the process of writing this thesis. Last but certainly not least, I want to thank my wife, *Ella*, for always believing in me. I could not have finished this without her.

Contents

1 Introduction

This thesis gives a basic introduction to machine learning and the workings of an artificial neural network. The focus is specifically on Convolutional Neural Networks applied to image recognition. The popularity of social networking and the practice of adding social tags to the things uploaded to different social networking sites have produced an enormous amount of data loosely labeled by the users themselves.

I implement and train a deep learning model to assess whether neural networks can be used with this kind of subjective image labeling to predict relevant social tags for a new image.

Firstly, I give a brief introduction to machine learning as a discipline, the relationship between biological learning and teaching a machine to learn, and discuss the motivation for developing deep learning models.

In the Chapter 2 the idea of an artificial neuron is introduced and I show how different variations of the concept can be used to construct a network similar to the structure of a human brain. The process of training the network through the use of gradient descent and backpropagation is presented. The problem of overfitting and ways to prevent it are discussed. The chapter ends with an examination of the workflow used for deep learning applications.

Chapter 3 focuses more specifically on Convolutional Neural Networks and their use in tasks related to image recognition. Relevant concepts such as convolution and pooling are introduced and I show how they are used to construct state-of-the-art models for image recognition.

In Chapter 4, I discuss the use of social tags on social networking sites. Some previous works on social tag prediction are also reviewed.

The rest of the chapters focus on describing the implementation, training and evaluation of a Convolutional Neural Network for social tag prediction.

First, I introduce two separate datasets consisting of images and related social tags uploaded to two different social networking sites (Chapter 6). I talk about preprocessing and pruning the data for training.

I give a brief introduction to a deep learning framework called Keras and show it can be used to construct a simple neural network (Chapter 7).

Next, in Chapter 8, I go through the process of adapting a pretrained network to be trained on the social tag datasets. I show specific implementation details with code segments and discuss the design decisions related to developing a neural network. I discuss training the network on the data and using the trained model to predict social tags for a new image.

In Chapter 9 the model is tested and the results are evaluated. I use statistical metrics and subjective evaluation to determine the quality of the predicted tags on a portion of the dataset reserved for testing.

The thesis ends with a short conclusion and a discussion on further research (Chapter 10).

2 Machine learning

Machine learning has become a ubiquitous part of life for most of us, whether we realize it or not. It is used for countless applications all around our daily lives. It is present in spam filters on our email accounts, the advertisements we see on our social media pages and it is used to help decide whether a bank will issue a loan. Machine learning combines ideas from neuroscience and biology with mathematical and statistical concepts.

Humans and other animals learn to adapt to new circumstances from experience. The crucial aspects of learning are *remembering*, *adapting* and *generalizing*. Remembering the last time you were in a similar situation, what action you tried and what the outcome was, you can adapt your plan of action. Machine learning is all about modeling those aspects of learning in computers. Instead of experience machines learn from data, and can be made to adapt their actions, such as making predictions, in a way that makes the predictions get more accurate over continued attempts.

Machine learning can be roughly classified into two categories based on the type of problem we are trying to solve:

- **Supervised learning** is the process of teaching a machine some features of *training examples* with known correct answers to produce the desired output with a new input. One example application of supervised learning is a junk mail filter trying to distinguish whether an email is spam or not.

- **Unsupervised learning** is used when we do not have the correct answers, or labels, for the input data. Instead, the algorithm attempts to identify features in the data and group similar items together. Unsupervised learning is used, for example, to discover which products have been frequently purchased together from a store's transaction data. [Marsland, 2015]

The main focus of this thesis is a branch of machine learning called *deep learning*, specifically *artificial neural networks*, which borrow concepts from the functionality of our own brains. Conventional computer programs are very good at quickly performing arithmetic computations and accurately following a list of instructions, both of which the human brain can stumble over. Conversely, some problems that humans can solve in microseconds, are utterly impossible for traditional programs. For example, a young child can instantly recognize whether a photograph is a picture of a dog or a cat, but the task is very hard to define in a set of instructions for a computer to follow. We could start by finding edges from the image and devise various rules about what kind of a shape constitutes a dog versus a cat. That approach could conceivably come up with a satisfactory model for recognizing cats and dogs through careful observation and countless rounds of trial and error. The problem arises when a new requirement turns up that we must also recognize rabbits and crocodiles. We must reconstruct the whole model to accommodate. Tackling these kinds of problems requires an entirely different way of programming a computer to mimic the human brain. [Buduma, 2017]

3 Artificial neural networks

Artificial neural networks first garnered interest after the publication of a paper by McCulloch and Pitts in 1943. They introduced a model of simplified artificial neurons as conceptual components for circuits capable of computational tasks [McCulloch and Pitts, 1943]. Since then, the study of artificial neural networks has advanced in strides and modern neural networks are often capable of matching or even outperforming humans in tasks that were once considered near impossible for computers.

The basic building block of artificial neural networks, the *neuron*, is based on the biological neurons found in the human brain. A simplified explanation of the current understanding of how we learn is that the neuron receives inputs through structures called dendrites, and the connections dynamically strengthen or weaken based on how often they are used. Each input connection's strength determines its weight when the inputs are summed and transformed in the cell body into a new signal which is the output of the neuron. This output is then transmitted to other neurons and the process continues. This practical understanding of biological neurons can be translated into a computational model. [Buduma, 2017]

The artificial neuron also gets some number of inputs, $x_1, x_2, \ldots, x_n$, which are multiplied by some weights, $w_1, w_2, \ldots, w_n$. The sum of those weighted inputs is called the *logit* of the neuron,

$$z = \sum_{i=0}^{n} w_i x_i \, .$$

This logit is used as the input for a function f to produce the output $y = f(z)$, as depicted in Figure 1. The inputs and weights are usually expressed in vector form $x = [x_1\ x_2 \ldots x_n]$ and $w = [w_1\ w_2 \ldots w_n]$ so the output can be computed as the dot product of the input and weight vectors $y = f(x \cdot w + b)$, where b is a constant bias.

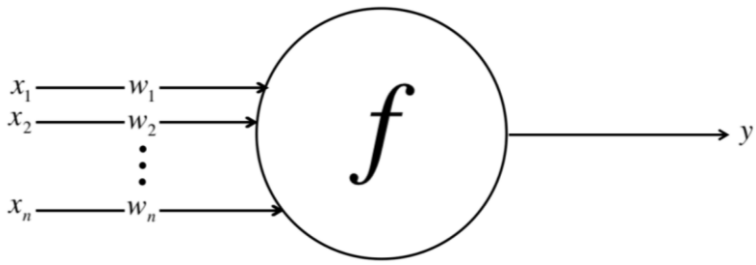

Figure 1. The schema for a neuron in an artificial neural net. [Buduma, 2017]

Creating an artificial neural network is connecting these neurons together to form an approximate model of how biological neurons connect to each other in the brain.

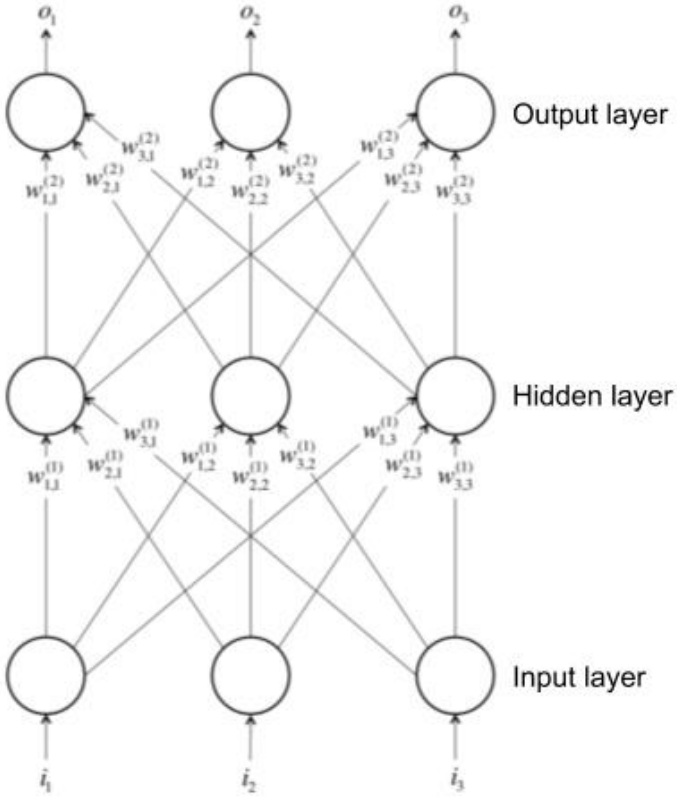

Figure 2. A simple example of a feed-forward neural network with three layers (input, one hidden, and output) and three neurons per layer. [Buduma, 2017]

Figure 2 depicts an example of a very simple network. The bottom layer of the network brings in the input data and the top layer computes the final result. This kind of network is called a feed-forward network, as there are no connections between neurons in the same layer or from a higher layer to a lower one. The layer in the middle, called the *hidden layer*, is what enables the network to learn and where most of the interesting work happens. Finding the optimal weights for the hidden layer is the answer to solving problems with neural networks. In more complicated neural networks there are usually multiple hidden layers with different numbers of neurons and each with their own weight vectors, connected to each other before feeding into the output layer. Often the hidden layers have far fewer neurons than the input layer which facilitates the learning of compressed representations of the input. As a biological analogy, our eyes receive a huge amount of raw input data through the photoreceptor cells, but our brain perceives it in terms of edges and contours. The hidden layers of the brain network come up with simplified representations of our surroundings.

As the neuron is mathematically expressed as a vector, we can express a neural network as a sequence of matrix operations. With an input layer $x = [x_1\ x_2 \dots x_n]$, we are trying to find the output vector $y = [y_1\ y_2 \dots y_m]$. This can be expressed as a matrix multiplication by constructing a weight matrix $W$of size $n \times m$ and a bias vector of length n. Each column of the matrix represents a neuron and the jth element of the column represents the weight of the connection. The transformation function

$$y = f(Wx^T + b),$$

is applied element-wise. [Buduma, 2017]

3.1 Neuron types

There are different kinds of neurons, usually defined by the function they apply to their logit. The most basic type of artificial neuron is called a *perceptron* [Rosenblatt, 1958]. It takes several binary inputs with weights and produces one binary output. Because of the binary nature of the perceptron, small changes to the weights of a single perceptron do nothing most of the time, but sometimes can make the output flip, for example from 0 to 1. This binary nature of the perceptron makes it hard to iteratively alter the weights to get closer to the desired result, which led to the development of neurons with inputs and outputs that can range on a scale. [Nielsen, 2015] The most simple type of these neurons uses a linear function of the form

$$f(z) = az + b.$$

This kind of neuron is simple to compute with, but it has some crucial limitations because any feed-forward network built with only linear neurons can be expressed as a network without any hidden layers. As stated before, hidden layers are what enables the network to learn the essential features of the input data. Consequently, neurons utilizing some kind of nonlinearity are necessary to learn complicated relationships. Three common types of neurons employing nonlinearity used in practice are the functions *sigmoid*, *tanh* and *ReLU*. [Buduma, 2017]

3.1.1 The sigmoid function

The sigmoid function is defined as

$$f(z) = \frac{1}{1+e^{-z}}.$$

When the logit z is small, the output is close to 0, and when the logit is large, the output is close to 1. Between the extremes, the function takes an S-shape, shown in Figure 3.

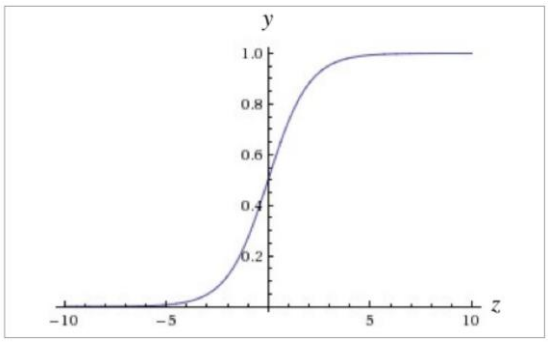

Figure 3. The output of a neuron using the sigmoid function as z varies. [Buduma, 2017]

3.1.2 The tanh function

The tanh function

$$f(z) = tanh(z)$$

produces a similar S-shaped nonlinearity, ranging from -1 to 1, shown in Figure 4. As the tanh function is zero-centered, it is often preferred over the sigmoid.

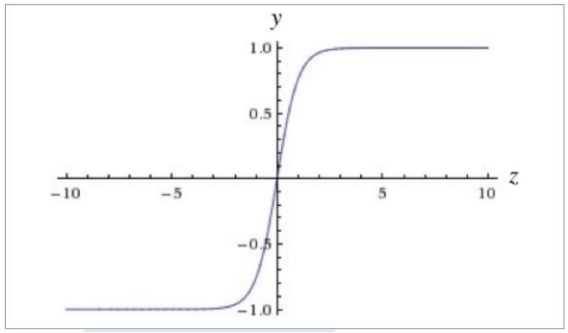

Figure 4. The output of a neuron using the tanh function as z varies. [Buduma, 2017]

3.1.3 The ReLU function

The rectified linear unit (ReLU) uses another kind of nonlinearity, by employing the function

$$f(z) = max(0, z).$$

This results in a hockey-stick-shaped output, shown in Figure 5.

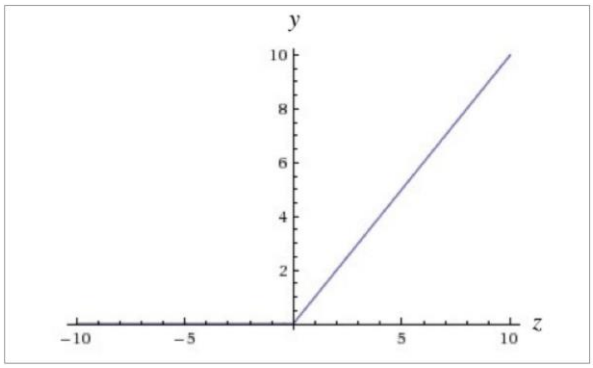

Figure 5. The output of a neuron using the ReLU function as z varies. [Buduma, 2017]

The ReLU function is widely used especially in applications of computer vision because it conserves more of the knowledge from the data during backpropagation than the S-curve functions. Using ReLU also leads to considerably shorter training times in large networks, as it does not involve computationally expensive operations like the tanh and sigmoid functions. [Krizhevsky et.al. 2012]

One drawback of ReLu is its fragility during training. A large gradient flowing through a neuron using the ReLu function can make the weights update in a way that "kills" the neuron, after which the neuron will not activate on any future data point. This can be controlled by a careful setting of a parameter called the *learning rate,* which I will discuss later in the thesis. [Karpathy, 2015]

3.1.4 The Softmax neuron

For classification tasks, it is helpful to have the output vector be a probability distribution over a set of mutually exclusive classes. For example, in the task of recognizing handwritten digits, the labels 0-9 are mutually exclusive. We can use a *softmax* layer to produce a probability distribution vector of the form $[\,p_1\ p_2 \ldots p_n]$ to provide a notion of how confident we are in the predictions. An output vector with one value close to 1 and the other values close to zero signifies a strong prediction, while an output vector with multiple values being close to equal signifies a weak prediction.

The softmax function is defined as

$$softmax(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}},$$

for $j = 1,2,\dots,K$. As the output of the softmax layer is a probability distribution, with the sum of the outputs being 1, the output of one neuron depends on the outputs of all the other neurons in the layer. [Buduma, 2017]

3.2 Training

The process of finding optimal values for the connection weights, the parameter vectors, is called training. The network is given large numbers of training examples and the weights are iteratively adjusted to minimize the output of a *cost function*, C, on the training data. A popular cost function is the mean squared error,

$$MSE = \frac{1}{2}\sum_i (t_i - y_i)^2,$$

where t_i is the target label for the ith training sample and y_i is the result given by the neural network. The output of the cost function is referred to as the cost or *error*. Other cost functions than mean squared error can also be used. The goal of training is to find the weights that result in a cost as close to 0 as possible. [Buduma, 2017]

3.3 Gradient descent

The method for finding the optimal weights is called *gradient descent.* In a simple case with a single neuron with only two inputs and thus two weights, $w_1$and w_2, the error space can be visualized in three dimensions, as shown in Figure 6. The vertical axis corresponds to the output of the cost function. [Buduma, 2017] In real applications there are often a considerably higher number of variables. Large neural networks can have billions of weights and biases which all affect the error at the output [Nielsen, 2015] .

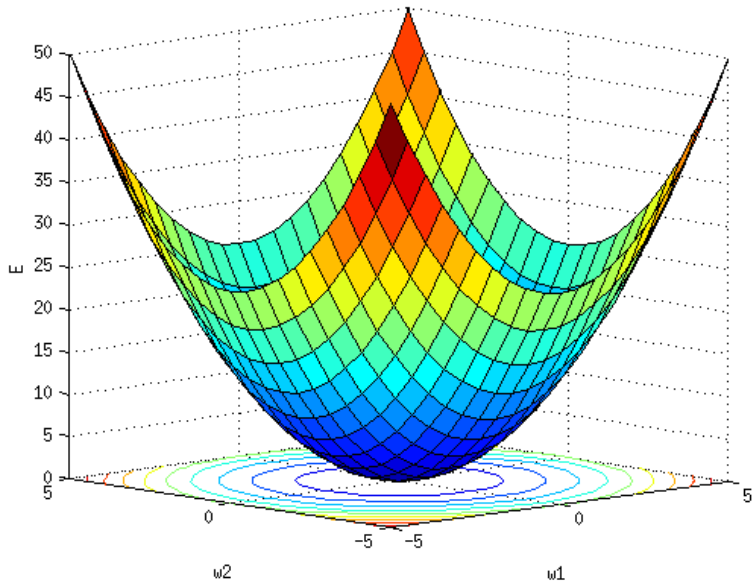

Figure 6. The error surface of a linear neuron with two input weights. [AI456, 2013]

Different settings of the two weights correspond to points on the horizontal plane, and the height is the error, or cost, at that point. Errors of all possible weights form a surface in the space, and the aim is finding the global minumum. For any point on the surface, the direction of the steepest descent is expressed as a vector called the *gradient*.

If the network is initialized with random values for the weights, the error will be somewhere on the surface. By iteratively determining the gradient and moving a step in the direction of the steepest descent by adjusting the weights, the cost will gradually decrease. The length of the step, or the amount of change in the weights, is determined by the steepness of the gradient. When the surface is more horizontal the minimum value is close, so the steps should be shorter. On a fairly level error surface, this can lead to the training taking very long, and to combat that, the gradient is often multiplied by a parameter called the *learning rate*.

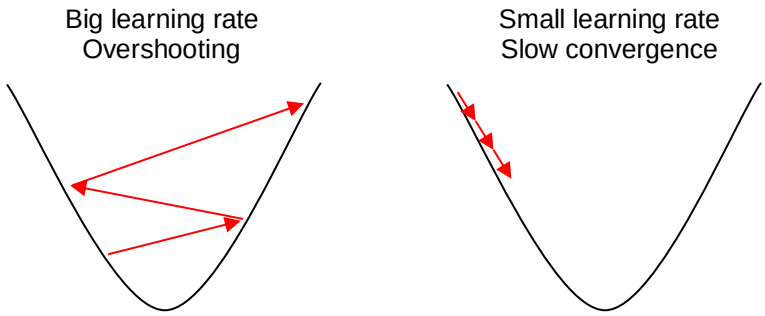

Figure 7. A too large or too small learning rate.

Picking an optimal learning rate is a hard problem on its own because a value that's too small will not solve the problem of slow convergence but an overly high value can cause the process to overshoot the minimum, as visualized in Figure 7. [Buduma, 2017]

There are other problems with this kind of *batch* gradient descent, where the whole dataset is used to compute the error surface. It works quite well with two weights, but usually the error surface is far more complicated.

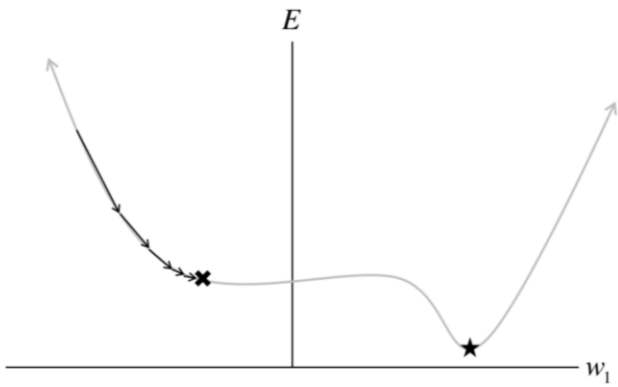

Figure 8. Batch gradient descent is sensitive to saddle points, which can lead to premature convergence. [Buduma, 2017]

Figure 8 depicts the error surface for a single weight. The surface has a flat region known as a *saddle point* where the randomly initialized gradient descent has settled, instead of finding the optimal global minimum. A way to deal with the problem is to compute the error surface for a subset of the whole dataset, called a *minibatch*, at each iteration, so instead of a single error surface, the surface is dynamic, as depicted in

Figure 9. This is known as *stochastic gradient descent* (SGD). In addition to helping avoid saddle points, stochastic gradient descent also leads to faster computation as the gradient vector is not calculated for the whole dataset at every step.

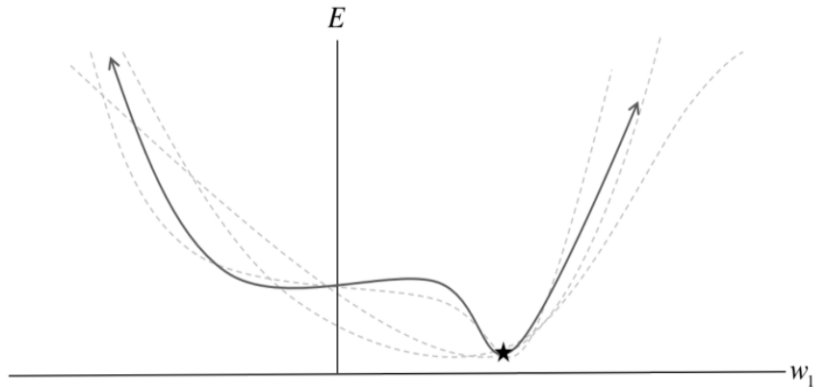

Figure 9. Dynamically fluctuating error surface. [Buduma, 2017]

Another problem with gradient descent is that the gradient does not often point out the optimal trajectory. The gradient consistently points toward the lowest point only when the contours are perfectly circular. With elliptical contours, the gradient can be up to 90 degrees off target. Figure 10 shows the actual directions to the lowest point (in blue) along a single step (in red).

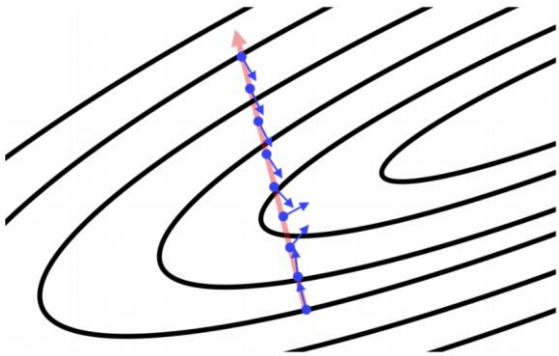

Figure 10. The direction of the gradient changes as we move along the direction of steepest descent. [Buduma, 2017]

A popular approach for dealing with fluctuating gradients is to introduce velocity-driven motion to the descent by the use of a *momentum* parameter. Every update to the weights is computed as a combination of the last update and the new gradient. This results in an exponentially weighted decay of past gradients and mimics the way a ball rolls down a hill. [Buduma, 2017]

Other efficient optimization algorithms built on stochastic gradient descent have been developed in recent years. One of the most popular is called *Adaptive Moment Estimation (*Adam*)*. Adam solves the problem of setting a suitable learning rate by calculating adaptive learning rates for individual weights and storing an exponentially decaying average of the past squared gradients, similar to momentum. Adam has been empirically shown to lead to faster convergence especially when used with large networks and training sets. [Kingma and Ba, 2014]

3.4 Backpropagation

Early research into artificial neural networks was hindered by the difficulty of computing the gradient vector used in gradient descent. Backward propagation of errors, or *backpropagation* for short, is a method for efficiently finding the gradient vector. Despite being invented in the early sixties [Bryson, 1961], it was only popularized in the late eighties after the publication of a paper by Rumelhart et al. describing a number of neural networks where the application of backpropagation makes them perform significantly faster compared to earlier approaches. The discovery made it possible for artificial neural networks to solve problems previously considered intractable. [Rumelhart *et al.*, 1986]

The aim of the backpropagation algorithm is understanding how altering the weights and biases in a network affects the cost. The weights are iteratively adjusted to optimize the cost by computing the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to a weight and a bias b. The basic idea is: For a particular training example, starting at the output layer, we can sum up the desired changes to the weights from the previous layer's neurons that will lead to a decrease in the cost, for each of the output layer's neurons. This essentially creates a list of "*nudges"* that we want to happen to the previous layers weights. Having those, we can recursively apply the same process to the previous layer all the way through the network. This process is repeated for every training example, to find the average *nudges* to each weight and bias in the network, and thus the gradient vector. [Nielsen, 2015]

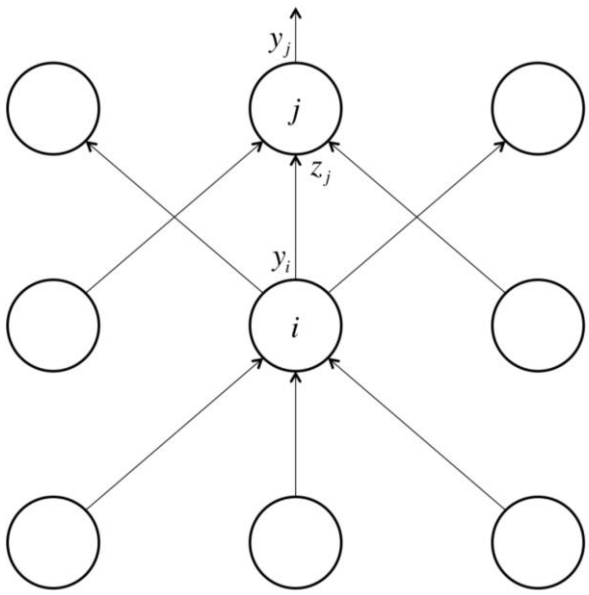

Figure 11. Reference diagram for the derivation of the backpropagation algorithm. [Buduma, 2017]

Referencing Figure 11, and assuming a sigmoid activation function, we start by calculating the cost function derivatives for the output layer j:

$$C = \frac{1}{2}\sum_{j \in output} \left(t_j - y_j\right)^2 \Rightarrow \frac{\partial C}{\partial y_j} = -\left(t_j - y_j\right).$$

To calculate the partial derivatives for the previous layer i, we can take advantage of the fact that the partial derivatives of the logit on the output layer j with respect to the connection from layer i is simply the weight of the connection w_{ij}:

$$\frac{\partial C}{\partial y_i} = \sum_j \frac{\partial C}{\partial z_j}\frac{\partial z_j}{\partial y_j} = \sum_j w_{ij}\frac{\partial C}{\partial z_j},$$

where z is the logit of the neuron. We can observe the following:

$$\frac{\partial C}{\partial z_j} = \frac{\partial C}{\partial y_j}\frac{\partial y_j}{\partial z_j} = y_j(1-y_j)\frac{\partial C}{\partial y_j}.$$

We can express the cost derivatives of layer i in terms of layer j cost derivatives:

$$\frac{\partial C}{\partial y_i} = \sum_j w_{ij}y_j(1-y_j)\frac{\partial C}{\partial y_j}.$$

Going through the training data, we can see how the weights affect the cost and determine how to alter the weights after each training sample:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial C}{\partial z_j} = y_i y_j(1-y_j)\frac{\partial C}{\partial y_j}.$$

Summing the partial derivatives over the training data gives us the formula for computing the list of *nudges* to apply to the weights:

$$\Delta w_{ij} = -\sum_{k\in dataset} \epsilon y_i^{(k)} y_j^{(k)}(1-y_j^{(k)})\frac{\partial C^{(k)}}{\partial y_j^{(k)}},$$

where ϵ is the learning rate. This process lets us influence the hidden neurons through observing how the weights affect them by looking at how fast the error changes when an individual weight is altered. [Buduma, 2017]

3.5 Overfitting

As is true for many machine learning approaches, building a very complex model can easily lead to perfectly fitting the training data. When such a model is evaluated on new data, it performs poorly. Overfitting is a huge challenge in any machine learning task, and especially so in deep learning as neural networks usually have many layers with a large number of neurons, producing a very complex model. One approach to prevent overfitting is dividing the training process into *epochs*, single iterations over the full training set. After each epoch, the model is evaluated on a set of validation data to see how well it is generalizing. If the accuracy keeps increasing on the training data, but stalls or decreases on the validation data, it is a sign of overfitting and the training should be stopped. Figure 12 shows an example division of the full dataset. [Buduma, 2017]

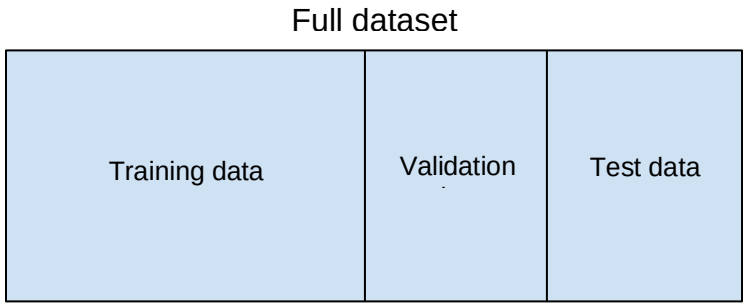

Figure 12. Division of dataset for training.

There are also several other countermeasures to avoid overfitting used in neural networks, that I will discuss next.

3.5.1 Regularization

The process of *regularization* tries to prevent overfitting by modifying the cost function with added terms that penalize large weights, so the cost function becomes $C + \lambda f(\theta)$, where $f(\theta)$ increases as the components of the weight vector θ grow, and λ is a parameter for the regularization strength. A small value of λ results in weaker protection against overfitting, but a value that is too high will prioritize keeping θ small over finding the optimal weights. [Buduma, 2017]

One type of regularization is called *L1 regularization* and it works by adding $\lambda|w|$ for every weight w to the cost function. This causes the weight vectors to become very sparse, leading the network to focus on a small number of the most critical inputs. This improves the resistance against noise in the inputs.

L2 regularization is another common approach, that greatly penalizes weight vectors with peaks as opposed to smoother vectors. In L2 regularization term $\frac{1}{2}\lambda w^2$ is added for every weight w, thus promoting the network to utilize all of the inputs moderately rather than focusing too much on just some of the inputs. L2 regularization is commonly referred to as *weight decay* because it causes the weights to linearly decay towards zero during gradient descent preventing the weights becoming too large.

L1 regularization can help develop an understanding of which features of the input data are contributing most, but L2 has been empirically found to perform better for most applications. [Buduma, 2017]

3.5.2 Max-norm constraints

The *max-norm constraint* approach also attempts to restrict the size of the weight vector, by setting an upper bound c on the magnitude of the vector and enforcing the constraint by projecting the vector onto an origin-centered ball of radius c. A useful characteristic of max-norm constraints is that as the weight updates are bounded, the network cannot *explode* even with a learning rate that is set too high. [Buduma, 2017]

3.5.3 Dropout

Dropout has recently become the preferred method for preventing overfitting [Buduma, 2017]. It works very differently compared to the previously mentioned approaches and

does not rely on modifying the cost function. Instead, dropout modifies the network itself, by temporarily disabling a random subset of the hidden neurons, as shown in Figure 13. [Srivastava *et al.*, 2014] Dropout is particularly effective at reducing overfitting in very large and deep networks, where overfitting is often severe [Nielsen, 2015].

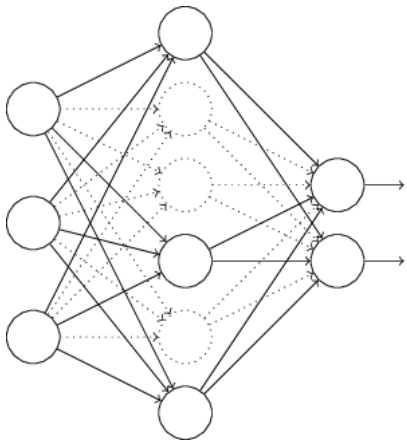

Figure 13. Half of the neurons in the hidden layer are disabled. [Nielsen, 2015]

The input is forward-propagated and the result backpropagated through the reshaped network. For the next iteration, the disabled neurons are restored and a new random subset is disabled and the cycle repeats itself. [Srivastava *et al.*, 2014] This results in a procedure that rather resembles training several different networks and averaging their results, without actually having to expend the resources to train a large number of complete networks. [Nielsen, 2015] The neurons in the network cannot rely on any other specific neurons existing, so they must learn features that are beneficial with many random groups of neurons, instead of forming relationships with some particular set of neurons [Krizhevsky *et al.*, 2012].

Because the weights and biases of the network are trained with only half of the neurons active during each iteration, the weights have to be halved when the full network is finally run. [Srivastava *et al.*, 2014]

3.6 Deep learning workflow

Similar other applications of supervised machine learning, the deep learning workflow usually starts with a thorough definition of the problem at hand. This consists of determining the vector forms of the inputs and potential outputs. [Buduma, 2017] For example, for recognizing handwritten digits from the MNIST database the input would be the pixel intensity values of the 28×28 pixel images, and the output layer would be a softmax with 10 neurons, one for each digit 0 through 9. Figure 14 shows a few examples from the MNIST database [LeCun *et al.*, 1999].

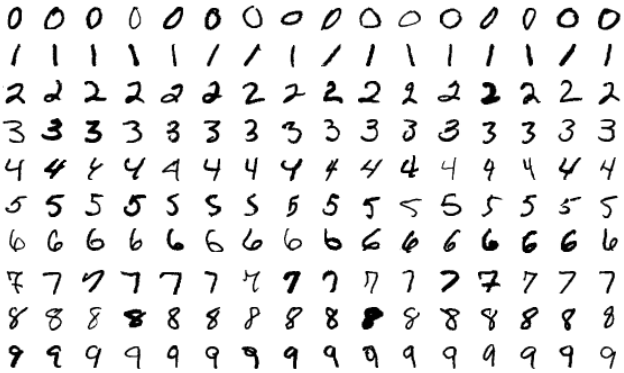

Figure 14. A few samples from the MNIST test dataset. [Steppan, 2017]

The internal structure of the network is defined next. The preferred architecture depends on the type of problem and includes deciding the number of hidden layers, the types of neurons, connections and so on. Different kinds of neural network architectures have been found to work well on different types of problems, for example, feed-forward *convolutional neural networks* are widely used for image recognition tasks [Krizhevsky *et al*., 2012], while *recurrent neural networks* work well for tasks involving natural language processing [Lai *et al*., 2015]. Figure 15 depicts a general workflow for creating any kind of neural network.

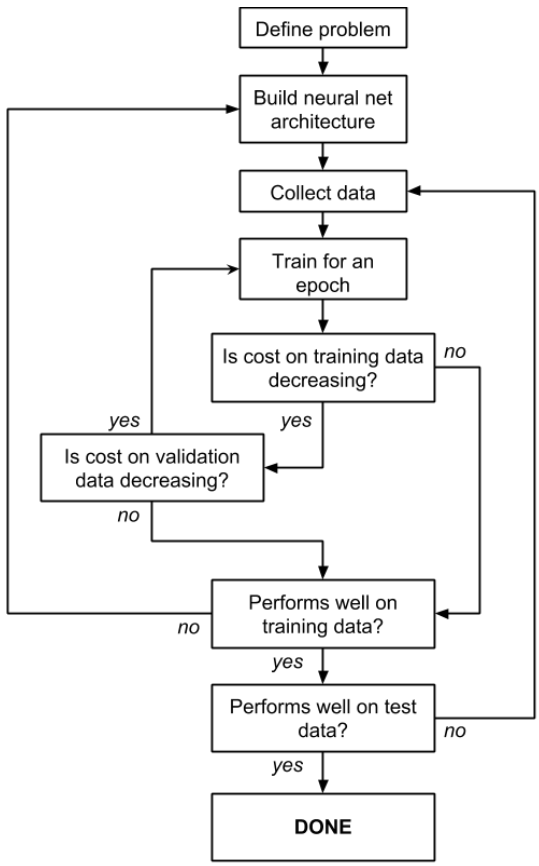

Figure 15. Detailed workflow for training and evaluating a deep learning model. [Buduma, 2017]

4 Convolutional Neural Networks

Image recognition is a great example of where neural networks perform well. The ImageNet challenge is a well-known benchmark in the field of computer vision. The task is to classify photographs into 200 categories with a training dataset of 450000 images. Before the advent of neural networks, the best approaches developed over years of research could not get an error rate under 25% [Perronnin *et al.*, 2010]. Krizhevsky et al. [2012] entered their method based on a *convolutional neural network* (CNN) into the competition in 2012 and achieved an error rate of 15.3% [ImageNet, 2012].

The task of image recognition proves to be too demanding for a regular neural network simply because of the sheer number of parameters. A small RGB image file of 300×300 pixels accumulate to a weight vector of length 270000. Adding several layers with full connectivity quickly adds up to an astronomical number of parameters. A convolutional network aims to constrain the number of connections to reduce the model's parameter count, by taking advantage of the 2D structure of an image. A CNN layer has neurons that are arranged in three dimensions. width, height, and depth. RGB images with three color channels have a depth of three, while width and height are simply the pixel dimensions of the image. Unlike in a regular fully-connected network, a neuron in a layer only connects to a small part of the previous layer, called a receptive field, as shown in Figure 16. [Karpathy, 2015]

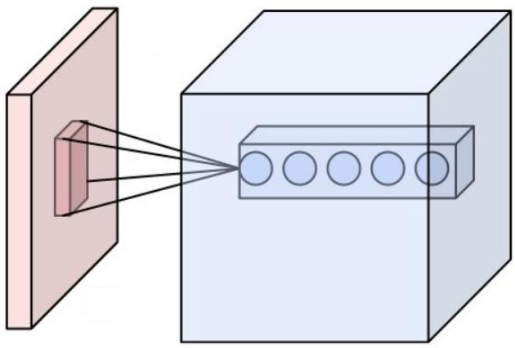

Figure 16. Neurons of a convolutional layer (blue), connected to their receptive field (red). [Aphex34, 2015]

There are three primary layer types used to build CNNs: *convolutional layers, ReLU layers, pooling layers and fully-connected layers.*

4.1 Convolutional layer

Convolutional layers do the majority of the computational work in a CNN. A CONV layer is built of filters, that are small along the width and height axes but preserve the depth of the input. With an input image of volume 300×300×3 and a receptive field size of 5×5, for example, neurons on the CONV layer will connect to a 5×5×3 region on the input, making up 75 weights. The receptive field slides through the width and height of the 300×300 input image and the neuron's filters compute dot products of the current position on the input and the filter values, producing an *activation map* for each filter. The number of filters and thus activation maps amount to the depth dimension of the output. A hyperparameter called *stride* determines how many pixels should the filter move over on each step. *Zero-padding* is another hyperparameter that allows

controlling the size of the output by adding zero values around the perimeter of the input. [Karpathy, 2015]

A filter can be considered a feature detector that attempts to identify simple details such as straight edges, curves or colors.

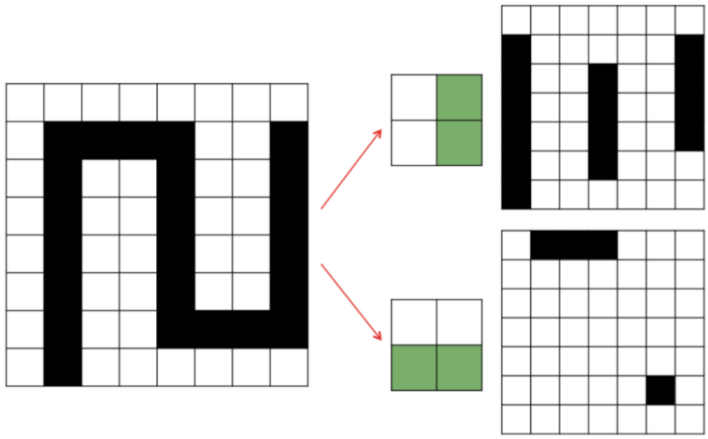

Figure 17. Applying filters that detect vertical and horizontal lines. [Buduma, 2017]

Figure 17 depicts a highly simplified example of applying filters to an image. On the left is an input image of dimensions 8×8×1 with vertical and horizontal lines. Two filters of size 2×2 (in green) are convolved over the picture with a stride 1, calculating an activation map by multiplying the pixel values of the current receptive field on the image with the filter values. The first one activates on horizontal lines and the second on vertical lines. Notice that the activation map's dimensions are 7×7, as there are that many positions for a 2×2 filter to fit on an 8×8 grid when moved 1 position at a time with no zero padding.

In this simple example there is only one color channel, black, and the input and filter values are binary, either 1 for black or 0 for white. On an RGB image, the pixel value ranges from 0 to 255 for each of the three color channels. [Karpathy, 2015] Figure 18 is a visualization of 96 filters learned by the 2012 winner of the ImageNet challenge. The input image dimensions were 224×224×3 and they used a stride of 4 pixels without zero-padding [Krizhevsky *et al.*, 2012].

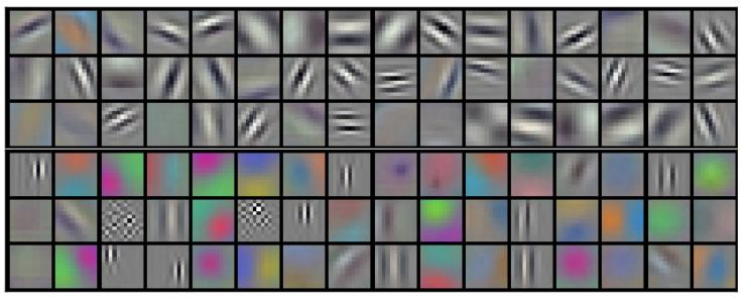

Figure 18. Filters of size 11×11×3. [Krizhevsky *et al.*, 2012]

It is common to use a ReLU layer after each convolution layer [Karpathy, 2015]. Krizhevsky et al. [2012] note that using ReLU instead of tanh makes the training time several times shorter, as the computational cost of ReLU is far lower than the nonlinear tanh.

4.2 Pooling layer

Convolutional layers are typically interspersed by pooling layers whose function is to scale down the width and height to cut down on the number of parameters. This not only decreases the computational burden of the network but also counteracts overfitting. The pooling filter is applied to each of the input's activation maps separately. A common pooling strategy is to use a 2×2 filter with a stride length of 2 performing a max operation, which selects the largest of the four values for the layer's activation map. The aim is to retain a fourth of previous layers activations that are the strongest. Figure 19 shows an example of activations produced by a 2×2 max pooling filter. [Karpathy, 2015] Using a spatially larger pooling filter leads to more information loss but a size of 4×4 may be warranted for very large input images in the first pooling steps to reduce the computational and memory load. Setting the stride length to be equal to the filter's width has been found to produce best results in most cases, but pooling can also be performed with overlapping receptive fields [Scherer *et al.*, 2010].

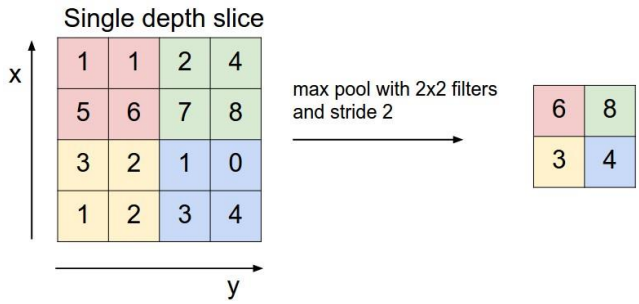

Figure 19. Max pooling. [Karpathy, 2015]

Other functions besides max can be used for pooling. *Average pooling*, which takes the average of the values in the receptive field has been commonly used in earlier research, but max pooling has been shown to produce better results in practice. [Scherer *et al.*, 2010] Some research supports not using pooling layers at all and instead decreasing the dimensions by using a larger stride on the convolutional layers [Springenberg *et al.*, 2014].

4.3 Fully-connected layer

Fully-connected (FC) layers are like the layers used in a regular, non-convolutional network. They connect their neurons to all of the outputs of the previous layer. FC layers are used at the end of the network to pull in the learnings of the convolutional layers and to produce a final output. After several rounds of convolution and pooling, the parameter size shrinks enough to make using fully-connected layers feasible. In a classification task, the final layer is typically a softmax layer that gives the probability distribution of the possible classes for the input image. [Karpathy, 2015]

4.4 Architecture of a CNN

A typical layer order of a CNN starts with pairs of convolutional layers combined with ReLU layers. The CONV-ReLU layer pairs can be stacked multiple times to feed into a max pooling layer. This whole structure may again be repeated several times before moving onto fully-connected layers, as shown in Figure 20. [Karpathy, 2015]

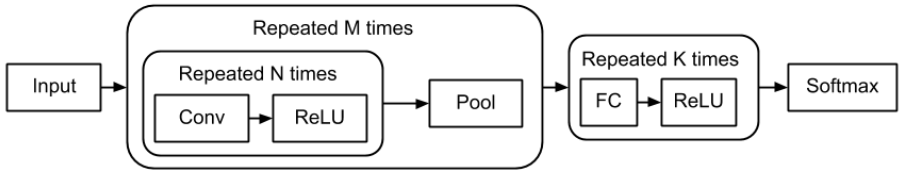

Figure 20. Typical architecture of a CNN.

Increasing the size of the network by adding more layers and filters is a simple way to get better performance. The drawback is that this obviously leads to a larger parameter count, increasing the network's computational load, while also making it more sensitive to overfitting. To combat this obstacle, researchers at Google developed a different way of building CNNs, dubbed the *Inception* architecture. The principal idea of the approach is to use multiple sizes of convolutions on the same layer and concatenate their outputs. For the implementation, Szegedy et al. [2014] chose convolution sizes 5×5, 3×3 and 1×1. They also included a max pooling layer of size 3×3 simply because pooling had historically been shown to be beneficial in CNN implementations. To further reduce the computational cost, the larger convolutions are preceded with a 1×1 convolution to reduce the size of the activation map.

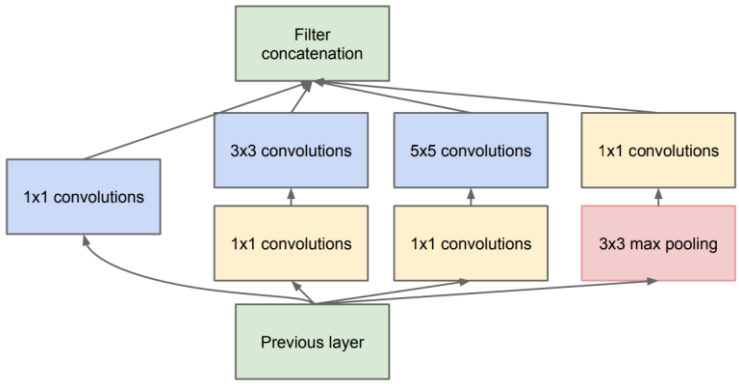

Figure 21. Inception module. [Szegedy *et al.*, 2014]

Figure 21 depicts a single layer, or *module*, of the Inception network. The network consists of these modules stacked together with an occasional max pooling layer in between. The authors found that having a few traditional convolution layers at the start of the network helped with memory efficiency during training. [Szegedy *et al.*, 2014] The team's Inception-based network, *GoogLeNet*, won the 2014 ImageNet image classification challenge with an error rate of 6.66% [Russakovsky *et al.*, 2015]. The architecture of the winning network is shown in Figure 22. Further research on the Inception architecture has been able to push the error rate even lower. The currently most recent revision, Inception-v4, boasts a 3.08% error rate [Szegedy *et al.*, 2016].

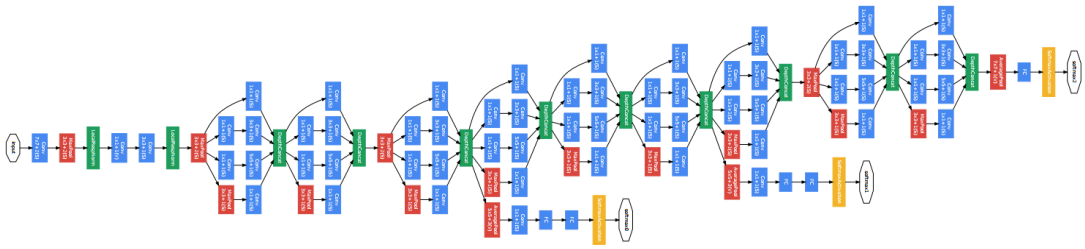

Figure 22. Architecture of the 2014 Imagenet challenge winner, GoogLeNet, with 9 inception modules. [Szegedy *et al*., 2014]

On the subject of designing the right CNN architecture for a practical application, Karpathy [2015] states: "*Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on ImageNet, download a pretrained model and finetune it on your data. You should rarely ever have to train a ConvNet from scratch or design one from scratch*".

5 Predicting social tags

In the context of the internet and social media, a *tag* is a keyword added to a piece of content, such as social media post. The author of the content can add tags to their post to allow other users to find content related to a theme by searching for a tag. On some sites, users can add tags not only to their own but also to other users' content. Social tags were popularized in the beginning of the 21st century by the social bookmarking site *del.icio.us* and the image sharing site *Flickr*. [Berendt and Hanser, 2007]

Today, most large social networking sites include some sort of tagging functionality. The most well-known implementation is perhaps Twitter's *hashtag*-system, where tags are preceded with the #-symbol. This scheme of using the #-symbol to denote a tag has been adopted by many other social media sites, such as *Facebook, Instagram,* and *Tumblr*. [Park *et al*., 2016]

A tag predictor can be used to enhance the tagging functionality of a social image sharing site by providing system suggestions for a new post and helping users agree on a vocabulary for certain subjects. [Heymann *et al*., 2008] It could also be used for image search, based on a keyword. A tag predictor can also help disambiguate between synonymous objects, like *Apple the company* vs. *apple the fruit* [Denton *et al*., 2015].

There has been some previous research on social tag prediction, but not very many approaches using the image pixel content as the source data for the prediction.

Heymann et al. [2008] used a web page's text to predict tags for a social bookmarking context using a support vector machine approach for the classification. While their work is not directly related to image recognition, there were insights that can be considered pertinent to social tag prediction in any context. For example, they found that the predictability of a tag is negatively correlated to its popularity.

Bertin-Mahieux et al. [2008] used a learning algorithm called *FilterBoost* to generate 360 different tags for songs to help aid with the cold-start problem of tag-based music recommender systems. The approach extracted acoustic features directly from MP3 files. The authors note that preprocessing the training data by removing tags with low classification rates could lead to better prediction performance. Their implementation ended up suggesting tags such as "*seen live*" and "*good music*", which may be meaningful for individual users but not very suitable for classification purposes.

Denton et al. [2015] were the first to use a convolutional neural network approach for predicting hashtags based on image content, as far as I know. They used a CNN that was pretrained on data from Facebook. Additionally, they experimented with embedding metadata about the user into the model. They hypothesized that additional information about the user, such as age, gender, and geographical location would help produce more accurate results. Their results do indeed show a significant boost to performance when user metadata was combined with the raw data. Unfortunately a dataset with this kind of user information is not accessible to me.

Park et al. [2016] also proposed a method using a CNN to predict hashtags based on image content. They used two pretrained CNNs, one specialized on objects and one on scenes, and combined their results to produce a multilabel classification of 1000 possible hashtags for images collected from Instagram. The authors found that achieving high accuracy metrics is difficult due to the highly contextual nature of user-

authored hashtags. The model may produce tags that seem to fit the image content, but do not match the original tags set by the user, as in Figure 23. I would argue that this is hardly a problem for the purposes of using the predictor for providing system suggestions for a new image.

Predicted Hashtag	GroundTruth Hashtag
food	work
yummy	tired
foodporn	
yum	
lunch	

Figure 23. Example of a challenging case. [Park *et al*., 2016]

6 Datasets

A suitable dataset is integral to the task of training and evaluating the performance of an artificial neural network. To implement a network capable of predicting social tags from image content, a dataset comprised of thousands of images and their accompanying tags is needed. Unfortunately finding a good quality dataset large enough proved to be a challenge. Ideally, I would prefer at least a few hundred thousand images from a social networking site like Instagram or Facebook. My first idea was to programmatically scrape the website *Instagram.com* and build my own dataset that way, but unfortunately Instagram's Terms of Use prohibit scraping [Instagram, 2013]. Because of this, I had to settle for two publicly available datasets, which each have their own drawbacks. The *HARRISON* dataset collected by Park et al. [2016] does consist of images downloaded from Instagram, but it is rather small at only 57383 images and 1000 tags. The *MIRFLICKR* dataset is larger, but it consists of images collected from a photography enthusiast website roughly ten years ago, so it does not exactly represent the tagging culture of modern social networking sites [Huiskes and Lew, 2008].

6.1 The HARRISON dataset

The *HAshtag Recommendation for Real-world Images in SOcial Networks* (HARRISON) dataset is a collection of 57383 images downloaded from the social image sharing site Instagram. The authors have already processed the related hashtags by lemmatizing words with the same base form. For example, *#walking* and *#walks* have been combined to the root word *#walk*. The dataset includes only 1000 of the most frequently used stemmed hashtags on the collected images. The average number of tags connected to one image is 4.5 (median 4, maximum 10). All images have at least one tag and each tag is connected to at least one image. [Park *et al*., 2016] The 10 most frequently used tags and their frequencies are listed in Table 1, along with the percentage of posts containing the tag.

Tag	Frequency	%-freq
love	5892	10.27%
friend	3646	6.35%
beach	3025	5.27%
family	2966	5.17%
yellow	2667	4.65%
girl	2614	4.56%
fashion	2371	4.13%
nike	2253	3.93%
snow	2212	3.85%
happy	2143	3.73%

Table 1. The most frequently used tags in the HARRISON dataset.

6.2 The MIRFLICKR dataset

The MIRFLICKR-1M dataset is a collection of 1 million images users have uploaded to the photography site Flickr, along with accompanying user tags. Flickr allows the uploader, and often also other users, to add tags to the images. The tags typically describe the subject of the image, acting basically as labels, but tags describing the context or environment the photograph was taken in are also frequent. The total number of unique tags in the dataset is 862114. The average number of tags connected to one image is 11.4 (median 9, maximum 75). There are 53887 images with no tags. The 10 most frequently used tags and their frequencies are listed in Table 2, along with the percentage of posts containing the tag.

Tag	Frequency	%-freq
nikon	45950	4.59%
canon	43308	4.33%
nature	40284	4.03%
2008	39955	4.00%
sky	33331	3.33%
blue	31930	3.19%
macro	30519	3.05%
bw	30290	3.03%
flower	29564	2.96%
water	28683	2.87%

Table 2. The most frequently used tags in the MIRFLICKR dataset.

All of the images in the collection are available under the Creative Commons license. [Huiskes and Lew, 2008]

As Flickr is a website aimed at photography enthusiasts, it is fairly common to use tags that refer to the camera equipment used to take the photo. As we can see from Table 3, two large camera manufacturers Nikon and Canon are the two most frequently used tags. These kinds of tags are not very useful for prediction, as they are not related to the content of the image, so we should *preprocess* the dataset to remove as many of them as we can. According to Ebay [2016] the 10 most popular camera manufacturers are (listed by popularity as tags):

Manufacturer	#	Frequency
Nikon	1st	45950
Canon	2nd	43308
Olympus	89th	9768
Sony	129th	7572
Pentax	131th	7553
Kodak	218th	4867
Leica	384th	2999
Fujifilm	386th	2950
Minolta	774th	1607
Samsung	1916th	727

Table 3. Popular camera manufacturers' frequencies in the tags.

Some popular camera models, such as *i500* and *d80*, and lens choices, like *50mm*, also appear fairly high on the list of frequent tags. Lenses and camera models typically include a number, so we can simply filter out tags containing a number. This also has the added benefit of discarding tags referring to the year the photo was taken, which is also a popular tagging practice that adds little advantage for the prediction task.

There are some obvious drawbacks to this kind of straightforward pruning of the tags. The targeted tags are not always unrelated to the image content. The tag *canon*, for example, could be used in connection to a photograph of a Canon camera, but this information is lost. Similarly, not all tags containing a number are necessarily irrelevant. Furthermore, while many camera models do contain a number, not all of them do, which leaves some of the models untouched. Nevertheless, I deem these shortcomings acceptable in order to eliminate a substantial amount of irrelevant information.

After removing the tags containing the name of a popular camera manufacturer or a number, we are left with 742250 unique tags. However, most of them (58.5%) are only used for a single image, making them unsuitable as training targets for a neural network. In fact, we probably only want tags that are connected to a decent number of images. If we remove tags that are connected to 100 or fewer images, we end up with 12234 tags, just 1.65% of the original tag count. The average number of tags connected to one image drops to 7.2 (median 8, maximum 72). There are now 89975 images with no tags, leaving us with 910025 images to be used as training examples.

7 The Keras library

Keras is an open source high-level neural network library written in the Python programming language. The library is not a stand-alone end-to-end machine learning framework, but instead can be run on top of several different lower level neural network libraries that Keras calls *backends*. Currently, three backend implementations are available for TensorFlow, Theano and, CNTK. The aim of Keras is to enable fast experimentation by having an easy to use API and a focus on modularity and extensibility. Keras was built to provide an intuitive set of high-level abstractions that make configuring neural networks simple regardless of the underlying computing library chosen.

Keras works around the idea of a *model* that comprises of *layers*. The simplest way to define a model is to use the Sequential model API and simply add a sequence of layers in the order they should be run. The library includes implementations of many types of fully connected and convolutional layers as well as pooling, dropout, and different activation layers. There are also recurrent layers for building recurrent neural networks and other specialized layer types. Keras also provides an API for implementing your own layers.

After defining the layers, the model is *compiled* which means configuring the learning process by defining:

1. The cost function, which Keras refers to as the *loss function*. Keras comes with several implementations of common cost functions such as MSE or *categorical cross entropy* (CCE), or you can implement your own.
2. The *optimizer* used in gradient descent. Common optimizers like SGD and Adam are included in the library.
3. A list of metrics used to judge the performance of the model. The metrics are not used directly in the training of the model, rather they are used for monitoring the training process and evaluating the model's effectiveness. Keras includes a few metrics such as *categorical accuracy* and *binary accuracy*. Custom metrics are also supported.

After compiling the model, it can be trained with the `model.fit()` -function, which takes the training data, the target data i.e. the labels, and some hyperparameters such as the number of epochs and batch size to be used during training. The function can also be given a set of validation data on which the model is evaluated after each epoch.

Code segment 1 shows the annotated code for creating a Keras model of a simple convolutional neural network that can be used for classifying the handwritten digits from the MNIST database. [Chollet, 2015]

```
# First we create the model object:
model = Sequential()
# Add first convolutional layer with 32 filters and a 3x3
# receptive field. The MNIST images are 28x28 pixels in
# black and white, so they have 1 color channel:
model.add(Conv2D(32, (3, 3), input_shape=(1, 28, 28)))
# Add activation layer:
model.add(Activation('relu'))
# Activation layers can also be specified with the conv layer:
model.add(Conv2D(64, (3, 3), activation='relu'))
# Add a max pool layer with a receptive field of size 2x2:
model.add(MaxPooling2D(pool_size=(2, 2)))
# Add a dropout layer that deactivates 25% of the inputs:
model.add(Dropout(0.25))
# Flatten the input shape to one dimension for the FC layer:
model.add(Flatten())
# Add fully connected layer with 128 outputs:
model.add(Dense(128, activation='relu'))
# Another dropout layer with 50% deactivation:
model.add(Dropout(0.5))
# Add final softmax output layer with 10 outputs for digits 0-9:
model.add(Dense(10, activation='softmax'))
# Compile the model with CCE loss and Adam optimizer:
model.compile(loss=categorical_crossentropy, optimizer=Adam(),
metrics=['accuracy'])
# Train the model for 20 epochs:
model.fit(train_dataset, train_labels, batch_size=128, epochs=20,
          validation_data=(validation_dataset, validation_labels))
```

Code segment 1. Example of a simple CNN in Keras.

When using very large datasets, loading all of the images into memory for training is not possible. For this purpose, Keras has a class called ImageDatagenerator, which enables reading the training data in batches from disk as needed. The API also includes capabilities for data augmentation which can help reduce overfitting. The augmentations include approaches such as random cropping, rotation and flipping the image along one axis. [Chollet, 2015]

8 Using Keras to predict social tags

Following the advice of Karpathy [2015], I decided to use an existing pretrained network that has been proven to provide good results. Fortunately, Keras already includes implementations and pretrained weights for a few networks that have done well in the Imagenet challenge. I chose the *VGG16* network designed by Simonyan and Zisserman [2014], because it has been shown to generalize well to other datasets. The network's weights have been pretrained on the ImageNet dataset. The convolutional part of the network is built from 5 sections, each containing two or three convolutional layers with 3×3 receptive fields and a max pooling layer of size 2×2 with a stride of 2. In the original design, the convolutional layers are followed by 4 fully connected layers and a softmax output layer but these have to be replaced in order to train the network to predict social tags. The structure is depicted in Figure 24.

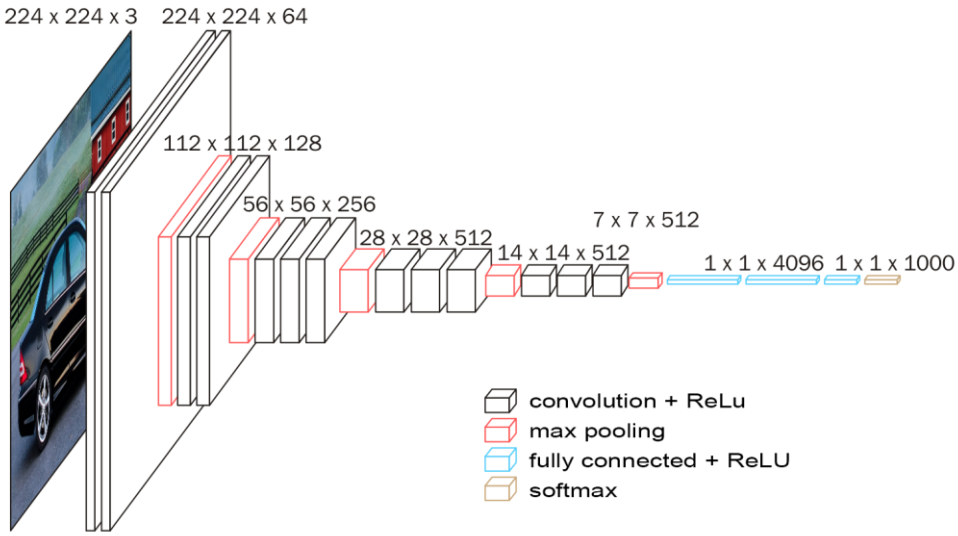

Figure 24. Structure of the VGG16 network. [Ahmed *et al.*, 2017]

In a normal image classification task where each image has one corresponding label, a softmax layer works well as the output. Predicting social tags is essentially a multi-label classification task as each image can have more than one tags. A sigmoid output layer, with a minimum threshold for the output, is more suitable for this purpose, as the output probability of each label is not affected by the other values, like in softmax.

The code for setting up the model for training is shown in Code segment 2. In the `setup_model` function, the pretrained model is loaded without the final fully connected layers, and new top layers are added.

```
def add_new_top_layers(base_model, nb_classes):
  x = base_model.output
  x = GlobalAveragePooling2D()(x)
  x = Dense(512, activation='relu')(x)
  x = Dropout(0.5)(x)

  predictions = Dense(nb_classes, activation='sigmoid')(x)
  model = Model(inputs=base_model.input, outputs=predictions)
  return model

def setup_model(nb_classes):
  # The argument include_top=False excludes the original FC layers
  base_model = VGG16(weights='imagenet', include_top=False)
  model = add_new_top_layers(base_model, nb_classes)
  return model
```

Code segment 2. Preparing a pretrained model for transfer learning.

8.1 Forming the target vectors

The ImageDataGenerator class has a method called `flow_from_directory` which generates the batches of training data and their corresponding target vectors. The function is designed for simple classification with one correct label and expects the image files to be placed in subdirectories, one subdirectory per class. An example of this kind of dataset consisting of images of dogs and cats is depicted in Figure 25. The array representation of a target vector for a picture of a dog would be `[1,0]` and `[0,1]` for a cat.

```
> tree train --filelimit=2
train
├── cat [12500 entries exceeds filelimit, not opening dir]
└── dog [12500 entries exceeds filelimit, not opening dir]

2 directories, 0 files
```

Figure 25. The directory structure for classifying images of cats and dogs.

This way of representing the labels by directory structure obviously will not work for a multi-label classifier, so some additional code is needed to complement the function. Instead of using the directory structure for determining the label, a `LabelEncoder` class reads the tags for each image from a CSV file and forms the target vector from those. The file consists of rows with the path to the image file and a space separated list of the tags related to that image. The target vectors are formed so that each tag has a corresponding index position in the array representation. If the image is linked to a tag, the value at that tag's position is 1, and if not it is 0. The resulting target arrays are very sparse as the number of possible tags is large, while single images are linked to few tags. An example array might look something like `[0,1,0,0,…,0,0,0,0,0,1,0]`. A new generator function called `multi_label_flow` is created using the `LabelEncoder` class to get the correct target vectors for the image files generated by `flow_from_directory`.

8.2 Data augmentation

Augmenting the training data by randomly warping the input images slightly has shown improvement to the classification performance of a CNN. It improves the robustness and accuracy of the network by effectively expanding the size of the training dataset. [Simard *et al.*, 2003; Fawzi *et al.*, 2016]. As mentioned before, the ImageDataGenerator enables randomly applying simple affine transformations to the images used for training the network. As a result, the network is never given the exact same image twice as an input, which helps reduce overfitting and improves the network's generalization ability.

The training data generator used is defined in Code segment 3.

```
train_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
  )
```

Code segment 3. Training data generator.

The parameters are:

- The `preprocess_input` function provided by Keras normalizes the input image by subtracting the mean RGB pixel intensity from each color channel.
- `rotation_range` specifies the maximum rotation in degrees
- `width_shift_range` and `height_shift_range` specify the fraction of the image dimension within which the input image is translated horizontally and vertically
- `shear_range` is the intensity of a counter-clockwise *shear mapping* in degrees
- `zoom_range` specifies the maximum zoom factor to use
- Setting `horizontal_flip=True` causes half of the images to be randomly flipped along the horizontal axis.

This kind of straightforward warping of the image applicable to ordinary photographs such as the user uploaded images from a social network, as the classification does not generally depend on the subject being in a specific orientation or scale.

8.3 Cost function

Choosing the right cost function is an important factor in designing a neural network. The chosen function affects how fast and if the network converges and whether it is able to find the global minimum cost during backpropagation. Keras includes many implementations of different cost functions or *loss functions* as they are called in the library documentation. Two popular cost functions included in Keras are *mean squared error* and *cross entropy*, which is defined for a single neuron as:

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)],$$

where n is the total number of training examples, a is the logit for a training example x, and y is the corresponding target output [Nielsen, 2015]. Golik et al. [2013] found that using *cross entropy* as the cost function leads to faster convergence than using *squared error*. Because of this a cross entropy based cost function was chosen. As the target vectors are very sparse, the model could achieve high accuracy by simply always predicting zero for every value of the output vector, i.e. predicting that the image has no tags. To remedy this, the values should be multiplied by a positive weight larger than 1 to decrease the false negative count and increase recall. Keras does not include this kind of cost function as part of the main library, but fortunately a function called `weighted_cross_entropy_with_logits` from the TensorFlow library can be adapted to work with Keras, as shown below in Code segment 4 [TensorFlow, 2018].

```
import tensorflow as tf
import keras.backend.tensorflow_backend as tfb

def weighted_binary_crossentropy(target, output):
    _epsilon = tfb._to_tensor(tfb.epsilon(), output.dtype.base_dtype)

    output = tf.clip_by_value(output, _epsilon, 1 - _epsilon)
    output = tf.log(output / (1 - output))

    loss = tf.nn.weighted_cross_entropy_with_logits(targets=target,
                                logits=output, pos_weight=10)

    return tf.reduce_mean(loss, axis=-1)
```

Code segment 4. Weighted cross entropy cost function.

Choosing the right weight for the function is an optimization problem on its own. For the network developed for this thesis, a weight of 10 was chosen arbitrarily. This seemed to produce acceptable results. Other hyperparameters for the network such as the learning rate, batch size, and dropout rate were also arbitrarily picked based on figures found in the literature and educated guessing. Optimizing the hyperparameter values could be a subject of further study beyond the scope of this thesis.

8.4 Transfer learning

The VGG16 model included in Keras is pretrained with ImageNet data, so it has already formed accurate activation maps and can be used to extract features from any image. This means the whole network does not need to be trained to work on the social image dataset. However, the newly added fully connected layers have randomly initialized weights and have to be trained on the fresh dataset. This can be achieved by *freezing* the convolutional layers of the network, so their weights are not affected by backpropagation, as shown in Code segment 5. The process is called transfer learning as it transfers learned features from one domain to another.

The first 19 layers of the VGG16 network comprise the convolutional part of it. Thus, calling the function below with `nb_layers_to_freeze=19` freezes all of the convolutional layers, by setting the `trainable` -property on the layer object to false, while leaving the final layers trainable. [Chollet, 2015]

```
def freeze_layers(model, nb_layers_to_freeze):
  for layer in model.layers[:nb_layers_to_freeze]:
     layer.trainable = False
  for layer in model.layers[nb_layers_to_freeze:]:
     layer.trainable = True
```

Code segment 5. Freezing the bottom layers of a network.

The pretrained weights of the model can also be adjusted to better fit the social image dataset by not freezing the convolutional part, or only freezing part of it, to allow backpropagation to also update the weights of convolutional layers. This process is referred to as *fine tuning*. Most convolutional neural networks that have been trained on image data tend to learn generic features resembling Gabor filters and color blobs in the first layers, regardless of the training dataset. These kind of universal feature detectors are broadly applicable to many other kinds of image datasets and benefit less from fine tuning. [Yosinski *et al.*, 2014] The deeper layers grow increasingly specific to the original training dataset and thus fine tuning some of the later convolutional layers could potentially increase the network's performance. Limiting the fine tuning to only part of the network also impedes overfitting and noticeably reduces training time as the number of trainable parameters is lower [Karpathy, 2015]. For the purposes of this thesis, I chose to freeze the first 16 layers of the network, leaving only the last 3 layers of the convolutional part to conform to the social image data. The social image data seem quite similar to the original ImageNet training data, so I believe most of the prelearned layers should perform reasonably well on the new dataset.

8.5 Optimizer

The Adam optimizer was selected because it has been shown to converge quickly and perform well at image recognition tasks. Adam is particularly effective on deep CNNs where the gradients on different types of layers vary greatly because it is able to adapt the learning rates for individual weights. [Kingma and Ba, 2014]. The initial learning rate was set to a relatively small value at 0.0001. The authors recommend a value of 0.001 which is also the default value in Keras but after testing, the lower value seemed to provide more reliable results while still converging in an acceptable time.

8.6 Training the network

Using the code segments presented above, a Python program called train.py was written to carry out the training of the network. The program takes 6 command line arguments which are explained below:

- `--dir` specifies the directory where the dataset is located. The program expects that directory to contain three subdirectories called `train` and `validation`, each of which in turn contain subdirectories of image files used for the corresponding task of the learning process. There is also a third subdirectory called `test`, which contains images to be used later in the evaluation of the trained network. The parent directory must also contain a CSV file called `tags.csv` which lists directory paths to each image and the social tags related to them.

- `--nb_epoch` sets the maximum number of epochs the training will continue if it is not halted before that by the user. This parameter is optional and defaults to 5.
- `--batch_size` sets the size of the batch used in the gradient descent. This parameter is optional and defaults to 32.
- `--output_model_file` specifies the file where the trained model file is saved after the training is finished. The parameter is optional and defaults to `my_model.model`.
- `--model` is an optional argument that expects a model file. This enables starting the training with a saved model with some pretrained weights instead of building the model from scratch each time. In a way, the training can be paused and continued, although the optimizer state is lost between instances.
- `--transfer_learn` takes a boolean value to determine whether the model should be trained with all the convolutional layers frozen.
- `--fine_tune` takes a boolean value to determine whether the model should be trained with the first 16 convolutional layers frozen.

8.6.1 Training on the HARRISON dataset

The model was trained on the HARRISON dataset for a total of 20 epochs. First 10 epochs with the transfer learning setup and then 10 epochs with the fine tuning setup. The model was saved after each epoch. The images were divided randomly into training, validation and test datasets. The training data comprised of 45908 images which represents 80% of the full dataset. Another 10% was used for validation during the training and the remaining 10% was reserved for evaluating the trained model.

The training was done on a 2015 model Apple MacBook Pro with a 2.5 GHz quad-core Intel i7 processor and 16 gigabytes of RAM as it was the fastest computer available to me at the time. Training the network took approximately 72 hours. The training duration could be significantly shorter if a more powerful computer with a modern graphics card were used.

The validation loss stopped decreasing after 18 epochs. This is a sign that the model was likely starting to overfit the data. At that point additional training would not increase the performance, so the model saved after the 17th epoch was used for testing and evaluation.

8.6.2 Training on the MIRFLICKR dataset

As the MIRFLICKR dataset is larger that the HARRISON dataset by an order of magnitude, training the network on a laptop computer proved unfeasible in a reasonable time frame. To speed up the training, a more powerful virtual machine was rented from a cloud computing provider, Paperspace.com. The training was done using a NVIDIA Quadro P4000 graphics processing unit, which significantly speeds up the training compared to running on a CPU.

Because of time constraints and the cost of running the rented machine, the network was only trained for 7 epochs on the MIRFLICKR dataset, 2 epochs with the transfer

learning setup and another 5 with the fine tuning setup. The training took approximately 65 hours. The validation loss was equal for the last two epochs, indicating that the training was probably reaching the end of its usefulness even with such few epochs.

The images were again divided randomly into training, validation and test datasets. The training data comprised of 728021 images which represents 80% of the full dataset. Again, 10% of the data was used for validation during the training and the remaining 10% was reserved for evaluating the trained model.

8.7 Predicting tags for new images

After training, the saved model can be used to predict social tags for images it has never seen before. Another Python program called predict.py was created for testing the trained model. It takes two command line arguments:

- `--dir` should be the same directory that was used for training the network. The images located in the `test` directory are used for testing and the tags are read from the `tags.csv` file as before.
- `--model` specifies the path to the saved model file.

The program uses the model to predict tags for each of the images located in the `test` directory. The predicted tags are defined by the output values of the final sigmoid layer. The values can be considered a measure of how *confident* the network is that a specific tag is associated with the image. The *confidence threshold* was set at 0.3, meaning that values over that are considered predictions. An upper bound was also set for the number of predictions for a single image. If there are many predictions with a confidence measure over the confidence threshold, only the 15 highest predictions are selected. The values for the confidence threshold and the maximum number of predicted tags were gathered empirically by running the network for a small number of images in the HARRISON dataset and examining the results of the sigmoid output layer. After numerous iterations, the prediction results were satisfactory with the chosen values.

9 Evaluation

The performance of the network was first evaluated by calculating accuracy metrics from the test datasets predicted tags.

As authors of previous work on social tag prediction have also found, achieving high accuracy metrics is difficult for a number of reasons, such as the subjective nature of social tagging and the high number of possible tags. However, while the absolute scores may be low, the overall subjective quality of the predicted social tags can still be reasonably good and relevant to the image content. [Park *et al*., 2016; Denton *et al*., 2015] For this reason, the results are also evaluated subjectively in addition to calculating statistical measures.

9.1 Accuracy metrics

Three metrics were chosen for evaluating the performance of the network: recall, precision, and the F_1 score. The metrics are calculated for each test image individually, using the predicted tags for that image and the known *true* tags of the image.

Recall is defined as

$$recall = \frac{number\ of\ correctly\ identified\ tags}{number\ of\ correctly\ identified\ tags + number\ of\ missed\ true\ tags}.$$

Precision is defined as

$$precision = \frac{number\ of\ correctly\ identified\ tags}{number\ of\ correctly\ identified\ tags + number\ of\ incorrectly\ identified\ tags}.$$

The F_1 score is the harmonic mean of the two, defined as

$$f1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

The mean value of each score is also calculated over the entire test dataset.

9.1.1 The HARRISON dataset

The model was tested on 5738 images from the HARRISON dataset which represents 10% of the whole dataset. The images used for evaluation were not used during training, so the network has never seen them before. Tags were predicted for each image and the precision, recall and F_1 scores calculated against the known true tags for that image. Table 4 displays the average values and standard deviation of the chosen metrics over the test images.

N = 5738	**Mean**	**Median**	**Std. deviation**
F1	0.07	0	0.11
Recall	0.15	0	0.25
Precision	0.05	0	0.08

Table 4. Average metrics of the HARRISON test data.

As is clearly visible, the resulting metrics are rather atrocious. The median value being 0, it is clear that the network mislabeled over half of the images. In fact, the model failed to predict a single correct tag on 3382 images, which represents 59% of the test set.

9.1.2 The MIRFLICKR dataset

The model was tested on 91001 images from the MIRFLICKR dataset which represents 10% of the whole dataset. Again, the images used for evaluation were not used during training. The evaluation was done the same way as before. Table 5 displays the average values and standard deviation of the chosen metrics over the test images.

N = 91001	**Mean**	**Median**	**Std. deviation**
F1	0.11	0.08	0.13
Recall	0.15	0.07	0.21
Precision	0.10	0.06	0.15

Table 5. Average metrics of the MIRFLICKR test data.

Again, the model performed quite poorly. On the MIRFLICKR data, the number of 0-scores was 42239, or 46% of the test set.

9.2 Subjective evaluation

Taking a closer look on the predictions reveals some insights into why the metrics were so low. Looking at the images with 0-scores, it seems that a considerable number of them have been labeled with predictions that appear to fit the image content fairly well but do not match the original tags added by the uploader. Figure 26 shows an image from the HARRISON dataset of a beach sunset. The predicted labels include tags like *nature*, *sunset*, *beach*, and *sea*, which I think can fairly objectively be considered appropriate tags for the image. However, the true tags do not reflect the image content, instead focusing on sort of *meta-tags* such as *instalike* and *photographer*, that the network did not pick up on, thus resulting in poor scores. Interestingly, the predictions do include the tag *photography* rather than the true tag *photographer*, which leads me to ponder whether further word stemming could be worth the lost nuance.

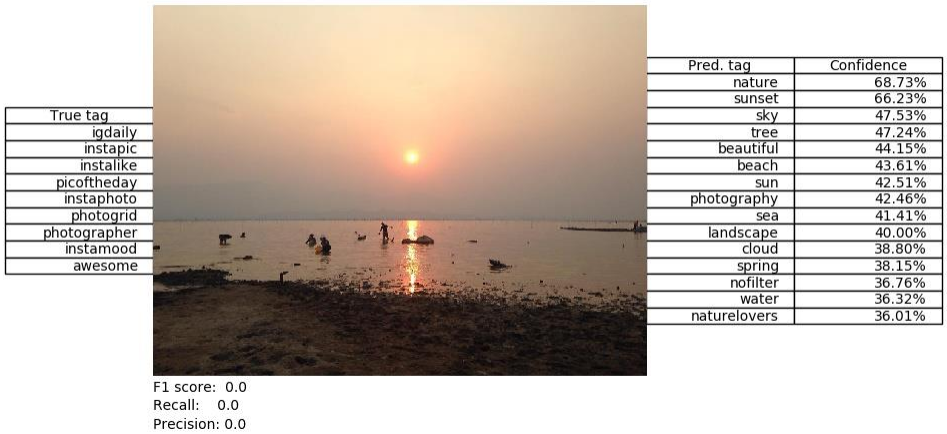

Figure 26. Example of incorrect tag predictions on the HARRISON dataset.

While there are certainly other reasons why the model produced poor predictions on a given image, this kind of pattern where the predicted tags are relevant to the image content, but the original tags are not, seems fairly common among the images that received poor metrics scores. I would argue that especially for the purpose of providing system suggestions for a new image, tags related to the image content would generally be the most relevant. This is why I think a subjective evaluation of the predictions is necessary to determine the model's performance and usefulness for such a purpose.

I enlisted the assistance of my wife as a test subject to help carry out the evaluation. We looked at 200 randomly selected images and their predicted tags from both datasets and counted the number of predicted tags that we personally considered to fit the image. This was divided by the number of all predicted tags, to get a subjective precision score, defined as

$$subjective\ precision\ = \frac{number\ of\ fitting\ predicted\ tags}{number\ of\ all\ predicted\ tags}\ .$$

For example, Figure 27 depicts another example from the HARRISON dataset where the network failed to predict the original tags, but the predicted tags seem to fit the image content.

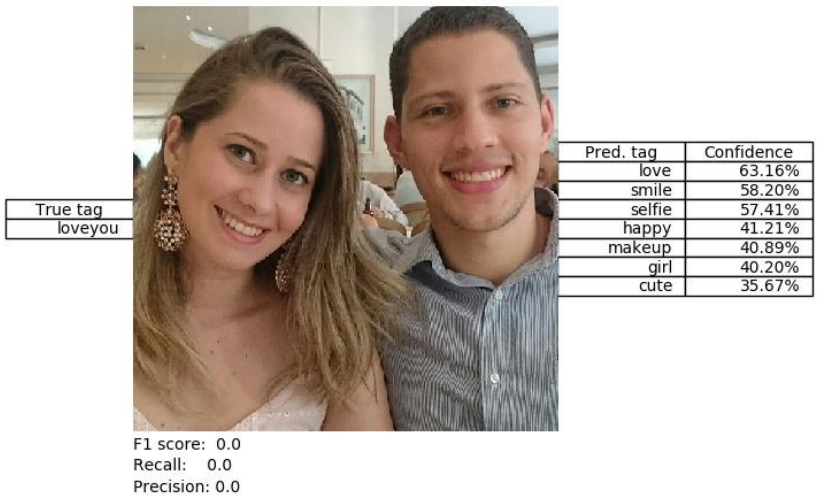

Figure 27. Predicted tags fit image content.

We judged all of the predicted tags to be relevant, giving this image a perfect subjective score of $\frac{7}{7} = 1$.

Surveying more than two people would quite obviously give more objective results, but because of constraints on time and resources, I deemed the selected method adequate for the purposes of this thesis.

9.2.1 The HARRISON dataset

We evaluated 200 randomly picked images from the HARRISON dataset. The average values of the scores are displayed in Table 6 and the frequencies in Figure 28.

Mean	Median	Std. deviation	N
0.57	0.55	0.28	200

Table 6. Average subjective scores on the HARRISON dataset.

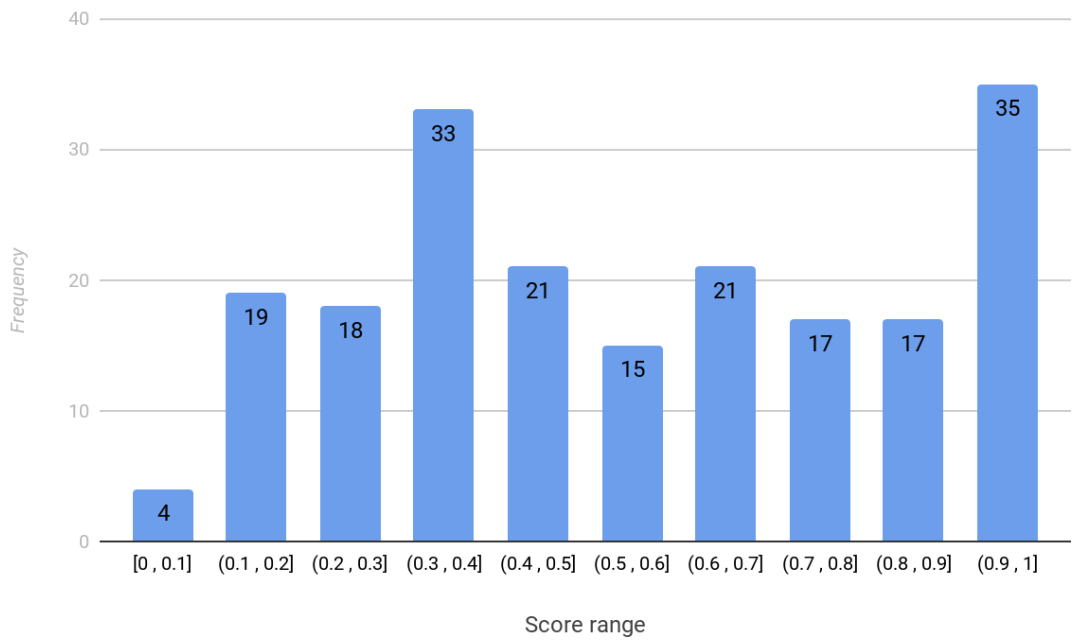

Figure 28. Frequency distribution of subjective scores on the HARRISON dataset.

As can be seen, the subjective evaluation delivered significantly better results than suggested by the metrics. 105 of the 200 evaluated images received a score greater than 0.5, and the network failed to predict a single fitting tag for only two of the evaluated images, meaning that 99% of the predictions contained at least one relevant tag.

Still, overall almost half of the predicted tags were not relevant. Particularly, tags related to food seemed to confuse the model. The network predicted tags such as *yummy*, *yum*, *foodporn* and *instafood* with a very high confidence for a large proportion of images that did not have anything to do with food. Figure 29 shows some examples of images the model seemed to think contained food.

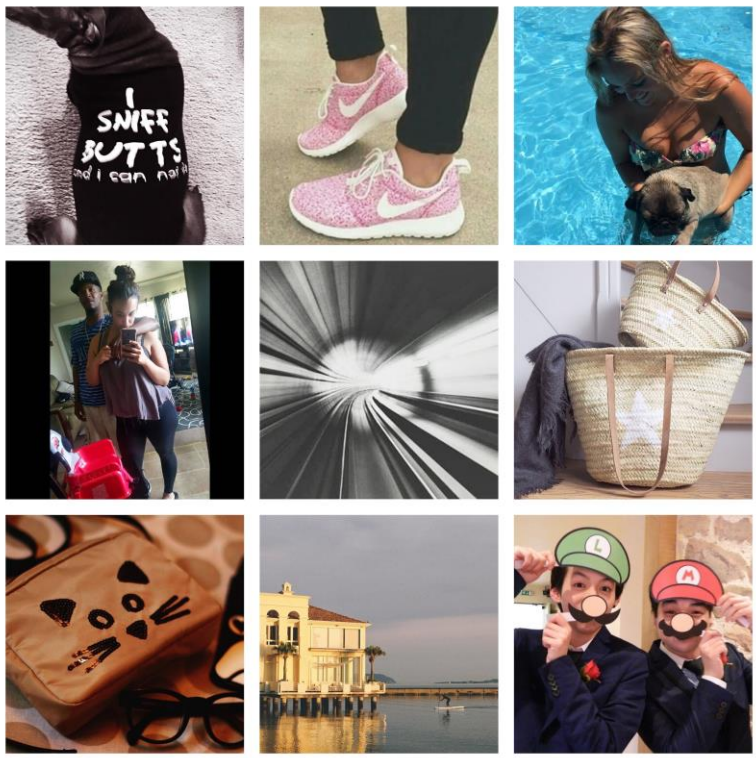

Figure 29. Examples of images the network confused with food.

We found that the model was especially good at recognizing landscape photographs and human faces, predicting tags such as *sea*, *sunset*, *nature*, and *landscape* or *selfie*, *cute*, *smile* and *girl* very reliably for relevant images.

9.2.2 The MIRFLICKR dataset

Again, we evaluated 200 randomly picked images from the MIRFLICKR dataset. The average values of the scores are displayed in Table 7 and the frequencies in Figure 30.

Mean	Median	Std. deviation	N
0.78	0.85	0.26	200

Table 7. Average subjective scores on the MIRFLICKR dataset.

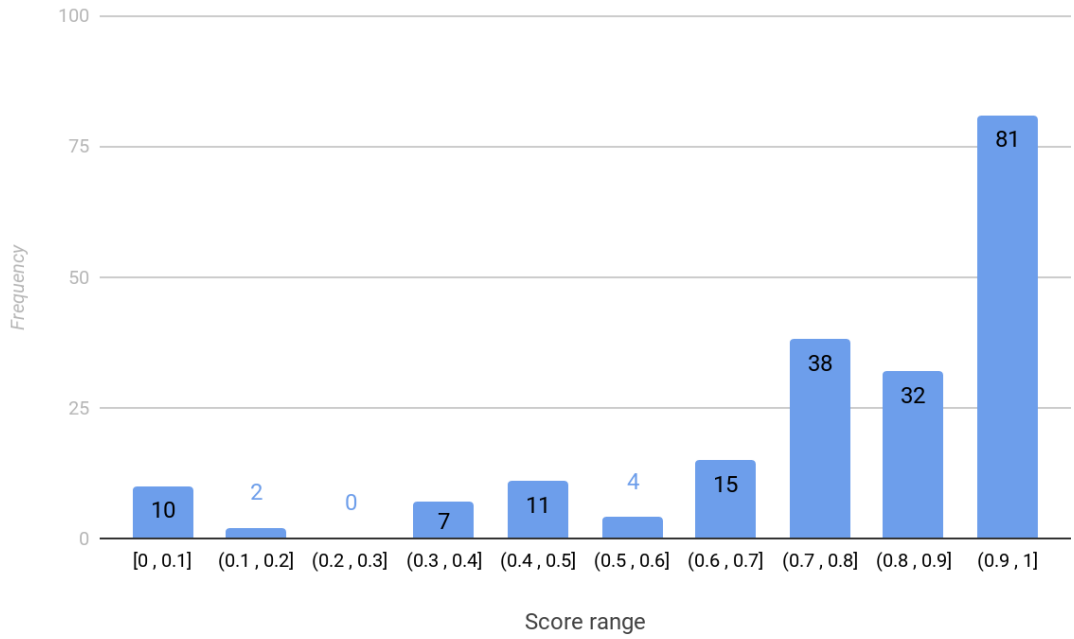

Figure 30. Frequency distribution of subjective scores on the MIRFLICKR dataset.

This time, the model seemed to perform even better, with over 40% of the images receiving scores higher than 0.9. The bulk of better performance is probably explained by the dataset's much larger size compared to the HARRISON dataset, and the tagging different tagging practices of the two social networks. The MIRFLICKR tags are frequently explicit labels of the image content, while on Instagram it is much more common for the tags to express the user's sentiments.

Many images, especially those depicting city skylines, graffiti, and other urban settings, are often tagged with the name of the location the image was taken. The network had trouble correctly predicting the actual location and instead tended to predict many conflicting tags for these kind of images. Figure 31. shows an example where the network predicted the tags *nyc*, *sanfrancisco*, and *london* for the same image of a graffito.

True tag
graf
graffiti
art
streetart
paint
painted
spraypaint
spraypainted
pets

Pred. tag	Confidence
graffiti	92.47%
streetart	81.64%
art	75.49%
street	57.61%
urban	41.19%
painting	38.89%
sanfrancisco	37.06%
nyc	36.52%
graf	35.96%
california	35.78%
graff	35.46%
paint	35.29%
london	34.25%
geotagged	31.62%

F1 score: 0.43
Recall: 0.55
Precision: 0.35

Figure 31. Multiple conflicting locations.

This is an example of a situation where having metadata, such as the GPS location of the photograph, included in the process could help produce more relevant tag suggestions. Determining the correct location based on image content alone is borderline impossible unless there are very many correctly tagged images of the same subject. For example, the network did manage to recognize and correctly tag pictures of the Eiffel tower with *paris*.

The dataset is multilingual and many of the predictions include translations of the same tag in multiple languages. Figure 32 shows an example where both the original tags and predictions contain the word for *cat* in many different languages.

True tag
cat
gata
kitty
gato
neko
kitten
black
fofurasfelinas

Pred. tag	Confidence
cat	92.91%
animal	65.32%
dog	61.34%
pet	60.54%
cats	54.17%
kitty	47.77%
animals	43.79%
gato	43.20%
white	41.25%
cute	38.49%
pets	37.74%
chat	33.32%
bestofcats	33.24%
black	33.21%
kitten	31.77%

F1 score: 0.43
Recall: 0.62
Precision: 0.33

Figure 32. Example of multilingual predictions.

Again, some metadata such as the GPS location or the user's language could help produce more relevant system suggestions. Another problem which metadata could alleviate is that some pictures were predicted to have many conflicting tags related to the time, such as a single image containing all of the seasons: *autumn*, *spring*, *summer*,

and *winter*. Having the time and date when the photograph was taken used as an input could improve the predictions.

The MIRFLICKR dataset does include the EXIF data for the images but the implementation is out of the scope of this thesis and is left for further research.

10 Discussion and conclusions

In this thesis I have examined the field of artificial neural networks. I have attempted to answer the question: *Can a Convolutional Neural Network be used to predict social tags based on image content?* The answer seems to be: *Yes, but including metadata would probably improve performance*. This application of CNNs has not been widely studied in the past and I find the insights gathered by my work to be interesting. However, the approach proposed in this thesis has certain limitations and would benefit from further research.

Denton et al. [2015] found that very frequently used tags tend to dominate the predictions for almost every image. This holds true for my results as well. The network does not see enough examples of the uncommon tags to be able to learn to predict them. Pruning the datasets by eliminating for example, the least frequently used 1% of the tags during preprocessing could have improved the accuracy of the predictions. Denton et al. experimented with using only the 10 000 most frequent tags and balancing their prevalence by downsampling the 500 most used tags to appear as frequently as the 501st, which enabled their network to better predict relevant but less frequently used tags. The same approach could be adapted to my work, but due to time restrictions it is left for future work.

Denton et al. [2015] also found that metadata about the user, such as age, gender and location helped produce more relevant predictions. Unfortunately a dataset containing this type of information is not available to me. As discussed in the previous chapter, the MIRFLICKR dataset does include another kind of metadata, i.e. the camera EXIF data, which could be helpful in producing more relevant predictions. Still, in this thesis I focused on using only the image pixel content as the input for the network and the implementation of including metadata is left for further research.

Another interesting research idea left out of the scope of this thesis would be to see if using a hybrid of a CNN and a traditional recommender system to provide the recommendations would result in more relevant system suggestions. Instead of directly training the network to predict social tags, the images could have small number of labels added, for example using the 1000 ImageNet [2012] categories. Then relevant tags could be found using a recommender system based approach that looks for tags used on similarly labeled images.

It is evident that the principle of using a CNN for social tag prediction is possible but more research is needed to reliably produce good results.

References

Saif Ahmed, Shams Ul Azeem, and Quan Hua. 2017. *Machine Learning with TensorFlow 1.x.* Packt Publishing.

AI456. 2013. Error surface of a linear neuron with two input weights, *Wikimedia Commons*, checked 1 October2018. <https://commons.wikimedia.org/wiki/File:Error_surface_of_a_linear_neuron_with_two_input_weights.png>.

Aphex34. 2015. Input volume connected to a convolutional layer, *Wikimedia Commons*, checked 1 October 2018. <https://commons.wikimedia.org/wiki/File:Conv_layer.png>.

Bettina Berendt and Christoph Hanser. 2007. Tags are not metadata, but "just more content" — to some people. In: *Proc. of the International Conference on Weblogs and Social Media*.

Thierry Bertin-Mahieux, Douglas Eck, Francois Maillet, and Paul Lamere. 2008. Autotagger: A model for predicting social tags from acoustic features on large music databases. *Journal of New Music Research* 37, 2, 115-135

Arthur E. Bryson. 1961. A gradient method for optimizing multi-stage allocation processes. In: *Proc. of the Harvard University Symposium on Digital Computers and Their Applications.*

Nikhil Buduma. 2017. *Fundamentals of Deep Learning*. O'Reilly.

François Chollet. 2015. *Keras*. GitHub, checked 1 October 2018. <https://github.com/keras-team/keras>.

Emily Denton, Jason Weston, Manohar Paluri, Lubomir Bourdev, and Rob Fergus. 2015. User conditional hashtag prediction for images. In: *Proc. of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1731-1740.

Ebay. 2016. Top 10 camera brands. Ebay, checked 1 October 2018. <http://www.ebay.com/gds/Top-10-Camera-Brands-/10000000205013205/g.html

Alhussein Fawzi, Horst Samulowitz, Deepak Turaga, and Pascal Frossard. 2016. Adaptive data augmentation for image classification. In: *Proc. of the IEEE International Conference on Image Processing (ICIP)*, 3688-3692.

Pavel Golik, Patrick Doetsch, and Hermann Ney. 2013. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In: *Proc. of INTERSPEECH*, 1756-1760.

Paul Heymann, Daniel Ramage, and Hector Garcia-Molina. 2008. Social tag prediction. In: *Proc. of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 531-538.

Mark J. Huiskes and Michael S. Lew. 2008. The MIR Flickr Retrieval Evaluation. In: *Proc. of the 2008 ACM International Conference on Multimedia Information Retrieval*.

ImageNet. 2012. Large scale visual recognition challenge 2012 results. ImageNet, checked 1 October 2018. <http://www.image-net.org/challenges/LSVRC/2012/results.html>.

Instagram. 2013. Terms of use. Instagram, checked 1 October 2018. <https://help.instagram.com/478745558852511>.

Andrej Karpathy. 2015. CS231n Convolutional neural networks for visual recognition. GitHub, checked 1 October 2018. <http://cs231n.github.io/>.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *Eprint arXiv:1412.6980*, checked 1 October 2018. <https://arxiv.org/abs/1412.6980>.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In: *Proc. of the 25th International Conference on Neural Information Processing Systems, Volume 1*, 1097-1105.

Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Recurrent convolutional neural networks for text classification. In: *Proc. of the 29th AAAI Conference on Artificial Intelligence*, 2267-2273.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1999. The MNIST database of handwritten digits, *MNIST database*, checked 1 October 2018. <http://yann.lecun.com/exdb/mnist/>.

Stephen Marsland. 2015. *Machine Learning: An Algorithmic Perspective, 2nd edition*. CRC Press.

Warren S. McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics* 5, 4, 115-133.

Michael A. Nielsen. 2015. *Neural Networks and Deep Learning*. Determination Press.

Minseok Park, Hanxiang Li, and Junmo Kim. 2016. HARRISON: A benchmark on HAshtag Recommendation for Real-world Images in Social Networks. *Eprint arXiv:1605.05054*, checked 1 October 2018. <https://arxiv.org/abs/1605.05054>.

Florent Perronnin, Jorge Sénchez and Yan Liu Xerox. 2010. Large-scale image categorization with explicit data embedding. In: *Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2297-2304.

Frank Rosenblatt. 1958. The Perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* 65, 386-408

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 533-536.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3, 211-252

Dominik Scherer, Andreas Müller, and Sven Behnke. 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In: *Proc. of the 20th International Conference on Artificial Neural Networks*, 92-101.

Patrice Y. Simard, Dave Steinkraus, and John C. Platt. 2003. Best practices for convolutional neural networks applied to visual document analysis. In: *Proc. of the 7th International Conference on Document Analysis and Recognition*, 958-963.

Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *Eprint arXiv:1409.1556,* checked 1 October 2018. <https://arxiv.org/abs/1409.1556>.

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *Eprint arXiv:1412.6806*, checked 1 October 2018. <https://arxiv.org/abs/1412.6806>.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1929-1958.

Josef Steppan. 2017. A few samples from the MNIST test dataset, *Wikimedia Commons*, checked 1 October 2018. <https://commons.wikimedia.org/wiki/File:MnistExamples.png>

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. 2016. Inception-v4, Inception-ResNet and the impact of residual connections on learning. *Eprint arXiv:1602.07261*, checked 1 October 2018. <https://arxiv.org/abs/1602.07261>.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. *Eprint arXiv:1409.4842*, checked 1 October 2018. <https://arxiv.org/abs/1409.4842>.

TensorFlow. 2018. API r1.6. TensorFlow, checked 1 October 2018. <https://www.tensorflow.org/api_docs/python/tf/nn/weighted_cross_entropy_with_logits>

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? In: *Proc. of the 27th International Conference on Neural Information Processing Systems, Volume* 2, 3320-3328.